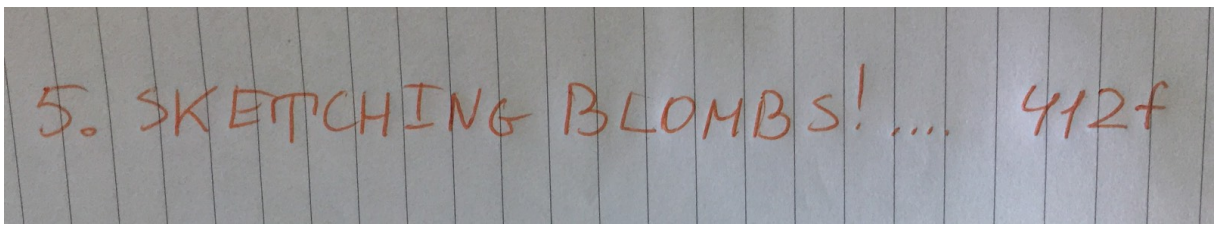
Making Fortran programs for solving an advection equation

Written by Evgenii Neumerzhitckii

Aug 31, 2019

Contents



1 The advection equation

In want to solve the following advection equation

$$v_t + vv_x = 0, \tag{1}$$

where v is velocity, t is time, x position, and v_t is a short notation for the derivative $\frac{\partial v}{\partial t}$. We have the following initial condition

$$v(x, 0) = \cos x, \tag{2}$$

and we want to solve the equation for values of x and t from the intervals

$$-\pi < x < \pi, \qquad 0 \leq t \leq 1.4. \tag{3}$$

2 Using analytical solution

Here we use analytical solution of Equation 1:

$$v = \cos(x - vt). \tag{4}$$

Our goal is to write a Fortran program that solves Equation 4 for various values of x and t parameters from the intervals given by Inequalities 3.

2.1 Using Newton-Raphson method

In order to solve Equation 4 numerically, we use Newton-Raphson method:

$$v_{n+1} = v_n - f(v_n, x, t)/f'(v_n, x, t), \tag{5}$$

where $n = 1, 2, \ldots, N_{max}$ is the iteration number with N_{max} being the maximum number of iterations, and

$$f(v_n, x, t) = \cos(x - v_n t) - v_n,$$

which is constructed by moving the terms of Equation 4 to one side. Here x and t are fixed values that do not change during this calculation.

We begin the calculations by choosing a starting v_1 value and then use Equation 5 to calculate v_2. Then we use v_2 to calculate v_3. This calculation is repeated until the absolute difference between two subsequent v values is smaller than a chosen tolerance number ϵ:

$$|v_{n+1} - v_n| < \epsilon.$$

The calculations are also stopped and the program is terminated with an error if the number of iterations exceeds a chosen maximum number of iterations N_{max}. The program is also terminated if devision by zero or an overflow is detected as a result of calculating v_{n+1} from Equation 5.

2.2 Finding roots for different values of x and t

Our goal is to find roots of Equation 4 for different values of x and t from the intervals defined by Inequalities 3. In order to calculate these roots, we use Newton-Raphson method multiple times by choosing values of x and t from the following sequences:

$$\{x_{\text{start}}, x_{\text{start}} + \Delta x, x_{\text{start}} + 2\Delta x, \ldots, x_{\text{end}}\}$$

$$\{t_{\text{start}}, t_{\text{start}} + \Delta t, t_{\text{start}} + 2\Delta t, \ldots, t_{\text{end}}\},$$

where x_{start}, x_{end} are the smallest and largest x values, and t_{start}, t_{end} are the smallest and largest t values that are supplied to the program by the user The values Δx, Δt are the position and time steps that are calculated as follows:

$$\Delta x = \frac{x_{\text{end}} - x_{\text{start}}}{n_x - 1} \tag{6}$$

$$\Delta t = \frac{t_{\text{end}} - t_{\text{start}}}{n_t - 1}, \tag{7}$$

where n_x and n_t are the number of position and time steps that are supplied to the program by the user.

2.3 Writing the code

Next, we write Fortran code to solve Equation 4. The code shown in Listing 1 is a part of `find_many_roots` function. The code calculates the roots of Equation 4 for various values of x and t parameters.

Listing 1: Solving $v = \cos(x - vt)$ equation for various values of parameters x and t (`root_finder.f90`).

```
160 ! Assign evenly spaced x and t values
161 call linspace(x_start, x_end, x_points)
162 call linspace(t_start, t_end, t_points)
163
164 ! Calculate step sizes
165 dx = (x_end - x_start) / (nx - 1)
166 dt = (t_end - t_start) / (nt - 1)
167
168 ! Calculate solutions for all values of x and t
169 do it = 1, nt
170     do ix = 1, nx
171         x = x_start + (ix - 1) * dx
172         t = t_start + (it - 1) * dt
173
174         root = find_root(options=options, x=x, t=t, success=success)
175
176         if (.not. success) then
177             ! Could not find root: return the problematic x and t
178             error_x = x
179             error_t = t
180             return
181         end if
182
183         solution(ix, it) = root
184     end do
185 end do
```

On `Line 161` we assign evenly spaced values between *x_start* and *x_end* for the x-coordinate and store then in the `x_points` array. The values `x_start` and `x_end`. The number of points *nx* are also supplied to the program by the user and they determine the size of the `x_points` array.

On `Line 162` use the same technique to calculate the values of the t-coordinate and store them in `t_points` array.

On `Lines 165` and `166` we calculate the position and time steps dx and dt using Equations 6 and 7.

Next, on `Lines 169` and `170` we use two loops to iterate over the range of indexes *it* and *ix*. Inside the loops, on `Lines 171` and `172` we calculate the values of the x and t parameters for the current iteration.

On `Line 174` we call `find_root` function. This function calculates a root of Equation 4 using Newton-Raphson method and returns this root.

Newton-Raphson method does not guarantee to find a root. The case when the program does not find a root is handled on `Lines 176-181`. Here we store the problematic values of x and t and exit the subroutine with an error.

Alternatively, in case when the program does find a root, we store that value in a 2D array

called `solution` on `Line 183`.

The end result of this part of the program is the `solution` array filled with values of v for all values of x and t parameters that the user has chosen. Alternatively, if the program could not find solution to Equation 4 for just a single pair of x and t, it exits with an error. In this case, the user is presented with an error message containing the values of x and t parameters for which Newton-Raphson method failed. The user can then re-run the program again with different values of starting v_1 value, tolerance and maximum number of iterations N_{max} for Newton-Raphson method, until she finds settings for which Newton-Raphson is able to find solutions for all values of x and t.

2.4 Plotting solution

We plot the output of our program Figure 1.

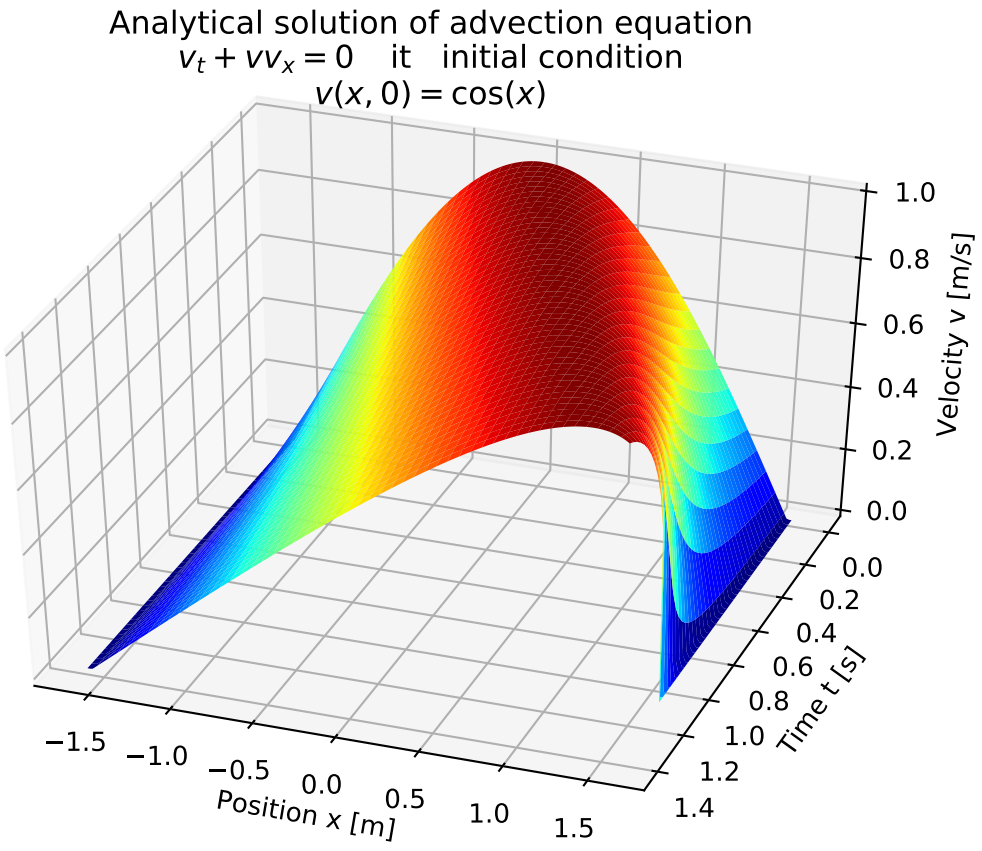


Figure 1: Analytical solution of advection equation.

The 2D plot of the solution is shown on Figure 2. We can see that as time increases, the solution curve becomes skewed to the right. After about $t > 1.2s$, the right side of the curve becomes almost vertical, breaks and does not reach the zero value.

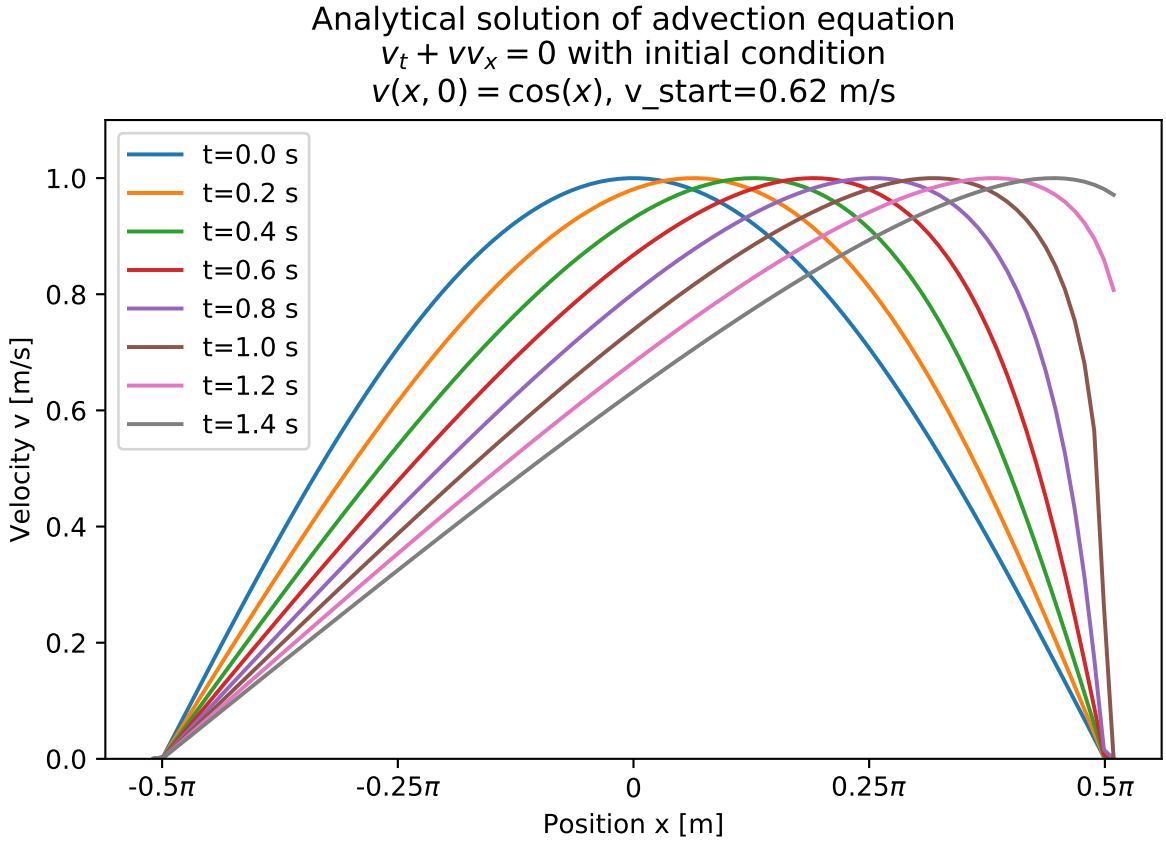


Figure 2: Analytical solution of advection equation for various time values. The solution was calculated using `v_start`=0.62 parameter for Newton-Raphson method.

In order to investigate the solution at high values of t we run the program with different values of `v_start` parameter (the initial value used for Newton-Raphson method), chosen between 0 and 1 We have found that Newton-Raphson method was not able to find a solution for most values `v_start` < 0.4. In contrast, the program was able to find solutions for values `v_start` > 0.5, especially those near `v_start` $= 0.8$.

In addition, we have found that the choice of `v_start` affected solutions for $t > 1.2$. For example, solutions calculated with `v_start` $= 0.15$ are shown in Figure 3. We can see that the right sides of the curves look different from those shown on Figure 2. It is possible that Equation 4 can be solved with multiple values of v for high values of t. Therefore, varying the value of `v_start` resulted in different values of v at the same x and t.

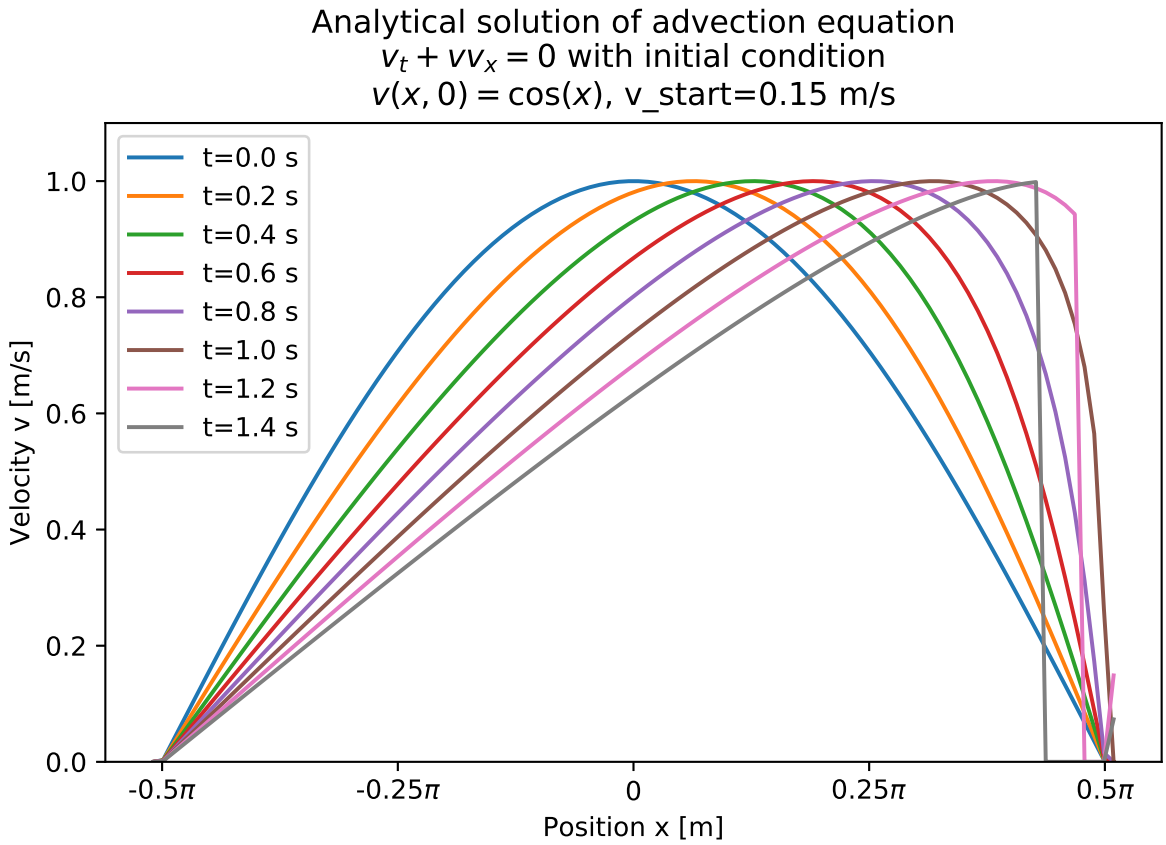


Figure 3: Analytical solution of advection equation for various time values. The solution was calculated using `v_start`=0.15 parameter for Newton-Raphson method.

3 Calculating numerical solution

We want to solve Equation 1 numerically. This can be done by first rearranging it into

$$v_t + (0.5v^2)_x = 0. \tag{8}$$

Next, we will use two methods of approximating this equation: centered-difference in space and an upwind method.

3.1 Centered-difference method

An approximate form of Equation 8 using forward-difference in space method is

$$v_j^{n+1} = v_j^n - \frac{1}{4}\frac{\Delta t}{\Delta x}\left[(v_{j+1}^n)^2 - (v_{j-1}^n)^2\right], \tag{9}$$

where the j is the position index and n is the time index:

$$j = 1, 2, \ldots, n_x$$

$$n = 1, 2, \ldots, n_t,$$

with n_x and n_t being the total number of position and time values respectively.

3.2 Upwind method

The second method we use for solving Equation 8 is the upwind method and it has the following recurrence relation:

$$v_j^{n+1} = \begin{cases} v_j^n - \frac{1}{2}\frac{\Delta t}{\Delta x}\left[(v_j^n)^2 - (v_{j-1}^n)^2\right], & \text{if } v_j^n \geq 0 \\ v_j^n - \frac{1}{2}\frac{\Delta t}{\Delta x}\left[(v_{j+1}^n)^2 - (v_j^n)^2\right], & \text{if } v_j^n < 0. \end{cases} \tag{10}$$

3.3 Writing the code

The Fortran code for solving Equation 1 using the centered-difference method is shown in Listing 2.

Listing 2: Solving advection equation (`advection_equation.f90`).

```
157 ! Assign evenly spaced x values
158 call linspace(x0, x1, x_points)
159 call linspace(t0, t1, t_points)
160
161 ! Set initial conditions
162 solution(:, 1) = cos(x_points)
163
164 ! Set boundary conditions
165 solution(1, :) = 0
166 solution(nx, :) = 0
167
168 ! Calculate the steps
169 dx = x_points(2) - x_points(1)
170 dt = t_points(2) - t_points(1)
171
172 select case (options%method)
173     case ("centered")
174         call solve_centered(nx=nx, nt=nt, dx=dx, dt=dt, solution=solution)
175     case ("upwind")
176         call solve_upwind(nx=nx, nt=nt, dx=dx, dt=dt, solution=solution)
177     case default
178         print "(a, a)", "ERROR: unknown method ", trim(options%method)
179         call exit(41)
180 end select
```

On `Line 158` we assign evenly spaced values for the x-coordinate and store then in the `x_points` array. The left and right boundaries `x0` and `x1`, as well as the number of the

values nx are supplied to the program by the user. Similarly, on `Line 159` we assign evenly spaced values in `t_points` for the time coordinate.

On `Line 162` we use the initial condition from Equation 2. Our velocity values are stored in a 2D array variable called `solution`. Here we assign the velocities to all the x-values corresponding to the first time index $n = 1$.

The boundaries of the position interval are supplied by the user and are assumed to be the values at which the cosine function is zero (the cosine function comes from Equation 2). For example, the left and right boundaries can be $-\pi/2$ and $\pi/2$. The full (i.e. not partial) time derivative of the velocity is

$$\frac{dv}{dt} = 0.$$

Therefore, if velocity is zero, it remains zero. We use this condition in `Lines 165` and `166` by setting zero velocities to the first and last position coordinate for all time indexes.

Next, on `Lines 169` and `170` we calculate the step sizes for position and time.

On `Lines 172-180` the program calls either the centered-difference (`solve_centered`) or the upwind subroutine (`solve_upwind`) based on the method chosen by the user.

3.3.1 Programing the centered-difference method

The body of `solve_centered` subroutine is shown in Listing 3.

Listing 3: Code for centered-difference method of solving advection equation (`advection_equation.f90`).

```
46 ! Pre-calculate the multiplier
47 a = 0.25_dp * dt / dx
48
49 ! Calculate numerical solution
50 do n = 1, nt - 1
51     solution(2 : nx - 1, n + 1) = solution(2 : nx - 1, n) &
52         - a * (solution(3 : nx, n)**2 - solution(1 : nx - 2, n)**2)
53 end do
```

On `Line 47` we calculate a multiplier that will be used later inside the loop.

On `Lines 50-53` we iterate over time indexes (n) and use the recurrence relation from Equation 9 to assign the values of velocities for the next time index (n + 1) using velocities that were calculated for the previous index of time (n). We assign the position indices in `solution` array by using the vectorised syntax

```
solution(2:nx-1, n+1) = ...
```

This allows to make calculations faster by executing multiple instructions in one CPU cycle using SIMD processor instructions.

3.3.2 Programing the upwind method

The implementation of the upwind method from Equation 10 is shown in Listing 4.

Listing 4: Implementation of the upwind method of solving advection equation (`advection_equation.f90`).

```
86  ! Pre-calculate the multiplier
87  a = 0.5_dp * dt / dx
88
89  ! Calculate numerical solution
90  do n = 1, nt - 1
91      do ix = 1, nx - 2
92          if (solution(ix + 1, n) > 0) then
93              q = 0
94          else
95              ! Add 1 to the x index if velocity is negative
96              q = 1
97          end if
98
99          solution(ix + 1, n + 1) = solution(ix + 1, n) &
100             - a * (solution(ix + 1 + q, n)**2 - solution(ix + q, n)**2)
101     end do
102 end do
```

The code structure is similar to that of the centered-difference method that we discussed. What's different herehere is that we need to use a second loop to iterate over the position indices (`Lines 91-100`) because the form of the recurrence relation equation depends on the sign of the velocity at each position step.

3.4 Plotting solutions

3.4.1 Plotting centered-difference solutions

We plot the output of the centered-difference program Figure 4. The corresponding 2D plot is shown on Figure 5. We can see that numerical solution closely resembles the analytical one from Figure 2 (analysis of errors is required to quantify the similarity between the two). However, it can be seen that numerical solution suffers from moderate jiggleschnook[1], that appears for large values of x at $t > 1.2$ s.

[1]Jiggleschnook is a technical term that means "a change of a quantity in such an unpredictable way that it looks almost idiotic". Naturally, jiggleschnook is a source of annoyance in science because it prevents describing a natural process with simple and aesthetically pleasing models. The word Jiggleschnook /'dʒɪg(ə)lʃnʊk/ is a combination of two words: "jiggle-" (English) meaning "to shake up and down or from side to side", and "schnook" (German), meaning "a fool". The term is said to be coined by a colleague of John Lattanzio.

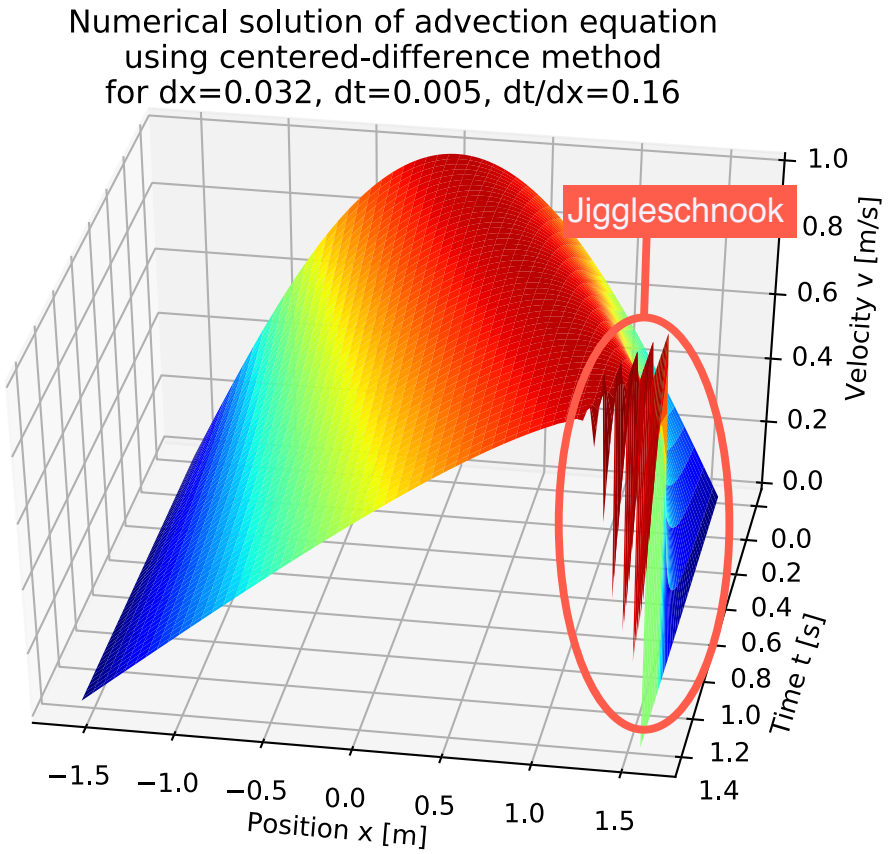


Figure 4: Numerical solution of advection equation calculated with centered-difference method using $\Delta t/\Delta x = 0.16\ s/m$. Moderate jiggleschnook can be seen at large position values for $t > 1.2\ s$.

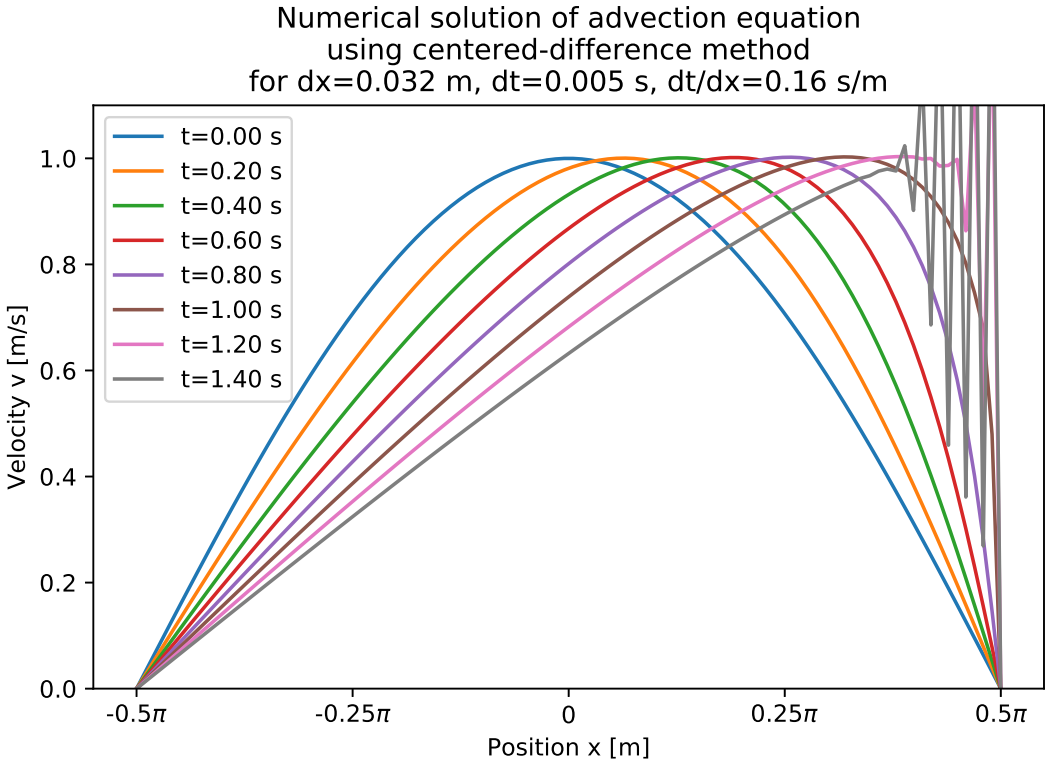


Figure 5: Numerical solution of advection equation calculated with centered-difference method using $\Delta t/\Delta x = 0.16\ s/m$. At $t = 1.2\ s$ a minor jiggleschnook appears at large position values, becoming larger at $t = 1.4\ s$.

The ratio of steps $\Delta t/\Delta x = 0.16\ sm^{-1}$ for the solution in Figure 5 is small. We want to investigate if solutions with higher $\Delta t/\Delta x$ show different amount of jiggleschook by solving the advection equation using $\Delta t/\Delta x = 1.0\ s/m$ (Figure 6) and $\Delta t/\Delta x = 1.59\ s/m$ (Figure 7).

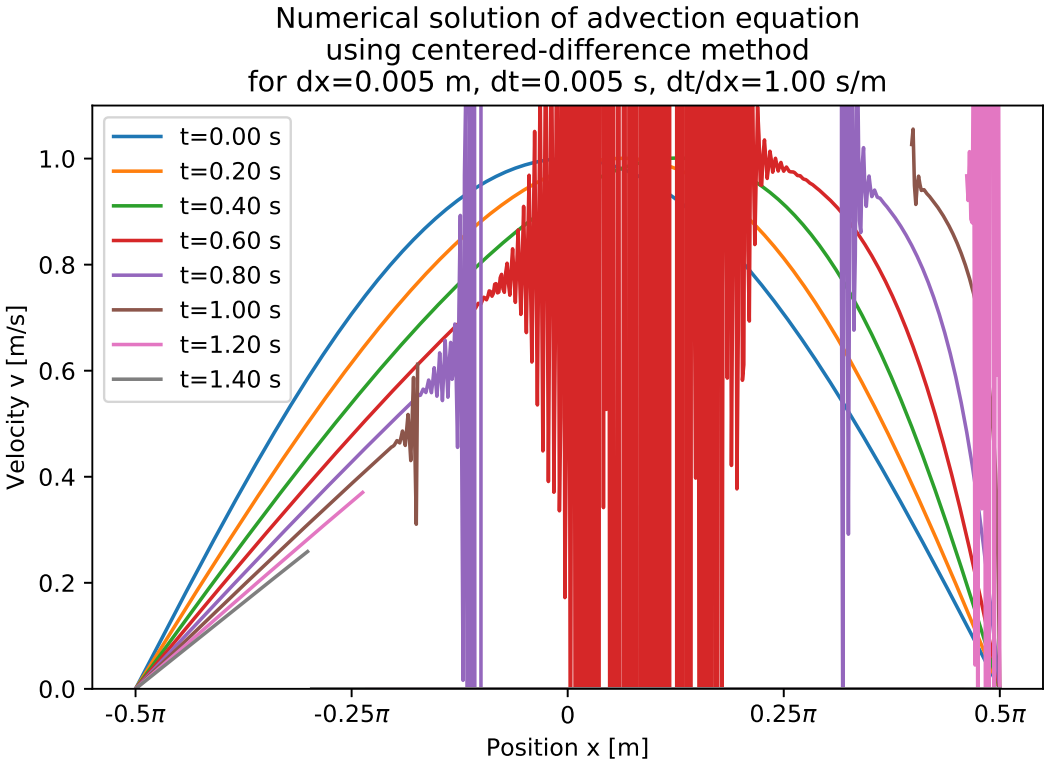


Figure 6: Numerical solution of advection equation calculated with centered-difference method using $\Delta t/\Delta x = 1.0\ s/m$. Strong jiggleschnook can be seen for $t > 0.4\ s$.

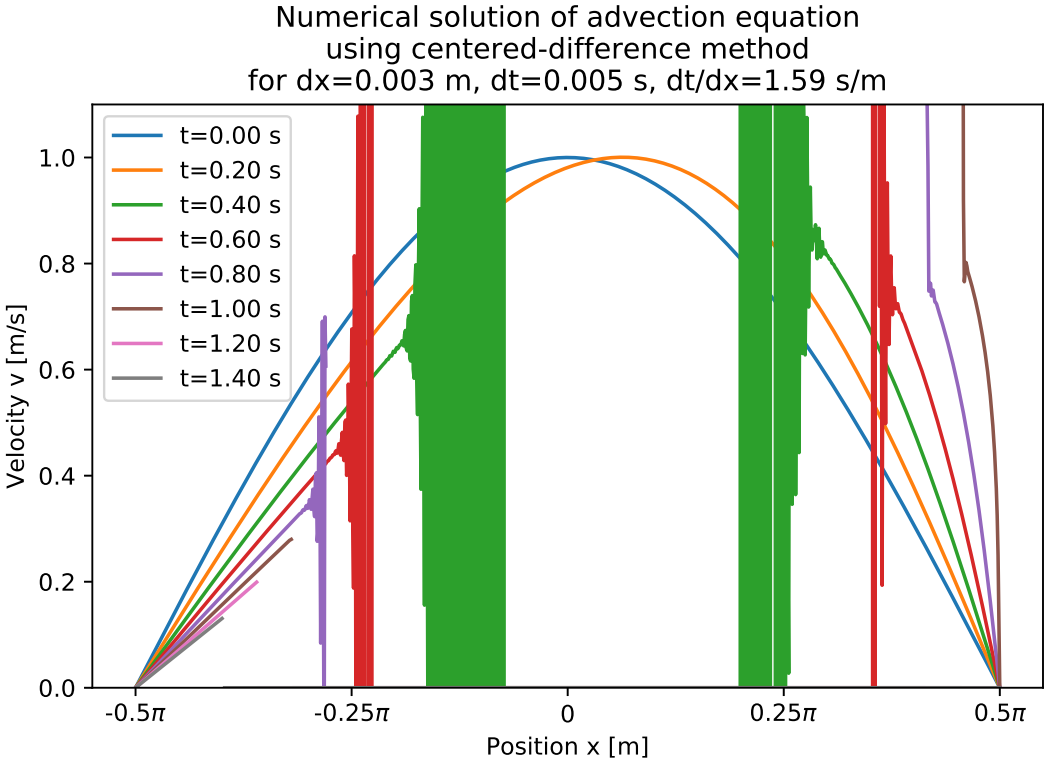


Figure 7: Numerical solution of advection equation calculated with centered-difference method using $\Delta t/\Delta x = 1.59\ s/m$. Solutions for $t > 0.2\ s$ are discontinuous and show very strong jiggleschnook.

It can be see from Figures 6 and 7 that increase of $\Delta t/\Delta x$ ratio results in the growth of jiggleschnook and causes discontinuities of solutions for higher values of t.

We conclude that our implementation of the centered-difference method can be unstable for values of $\Delta t/\Delta x$ both larger and smaller than 1 s/m, and that this implementation seems to show more instabilities for larger values of $\Delta t/\Delta x$. The sources of these instabilities can be bugs in our code, and/or numerical problems such as growth of errors due to rounding or catastrophic cancellation. Thus, we recommend a closer look at this code with the goal of finding and potentially fixing the instability issues.

It is also possible that centered-difference method used in our code is unstable simply by its very nature for any sizes of position and time steps, and thus the method is not suitable for solving Equation 1. A mathematical analysis of numerical stability of this method is required in order to support this claim.

3.4.2 Plotting upwind solutions

Next, we plot the solutions calculated with the upwind method for $\Delta t/\Delta x = 0.16$ s/m, shown on Figures 8 and 9.

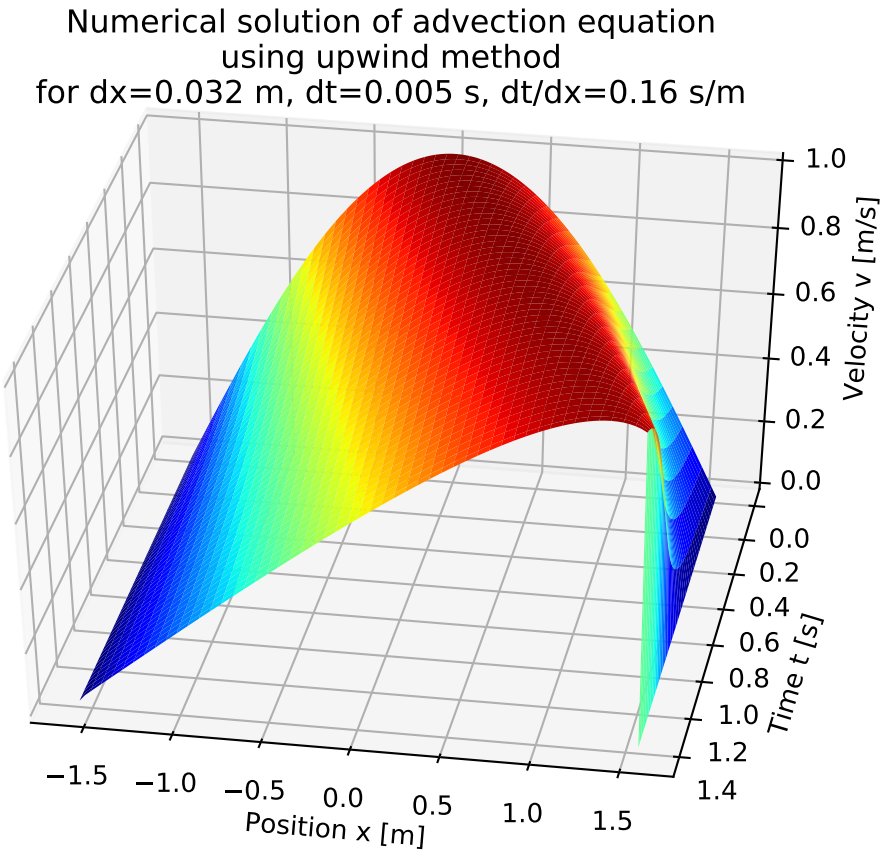


Figure 8: Numerical solution of advection equation calculated with upwind method using $\Delta t/\Delta x = 0.16$ s/m. The method appears to be stable for all values of x and t.

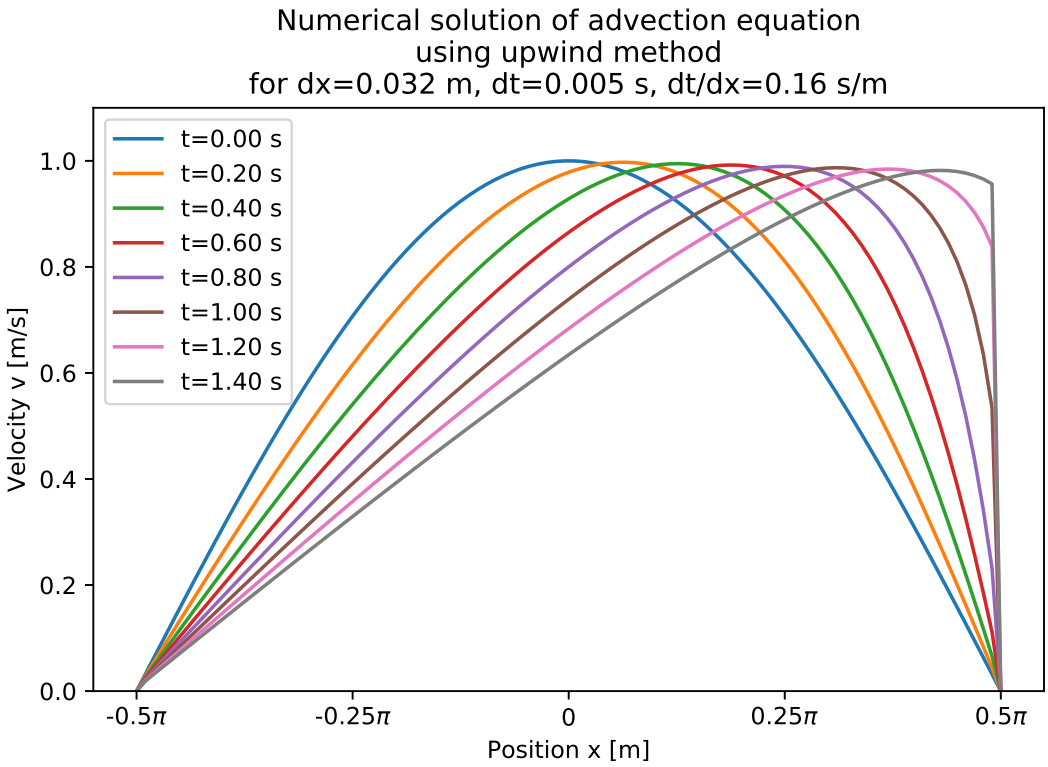


Figure 9: Numerical solution of advection equation calculated with upwind method using $\Delta t/\Delta x = 0.16\ s/m$. The method appears to be stable for all values of x and t.

We can compare the solutions for $\Delta t/\Delta x = 0.16\ s/m$ produced by the centered-difference (Figure 5) and upwind methods (Figure 9). We can clearly see that, unlike the centered-difference method, the upwind method appears to be stable for $\Delta t/\Delta x = 0.16\ s/m$. Better yet, solutions from the upwind method look more continuous and regular than even the analytical solutions shown in Figures 2 and 3.

Next, we want to investigate how the numerical stability of the upwind method depends on the $\Delta t/\Delta x$ ratio. Solutions for $\Delta t/\Delta x = 1\,s/m$ are plotted on Figure 10 and $\Delta t/\Delta x = 1.08\,s/m$ solutions are shown on Figure 11.

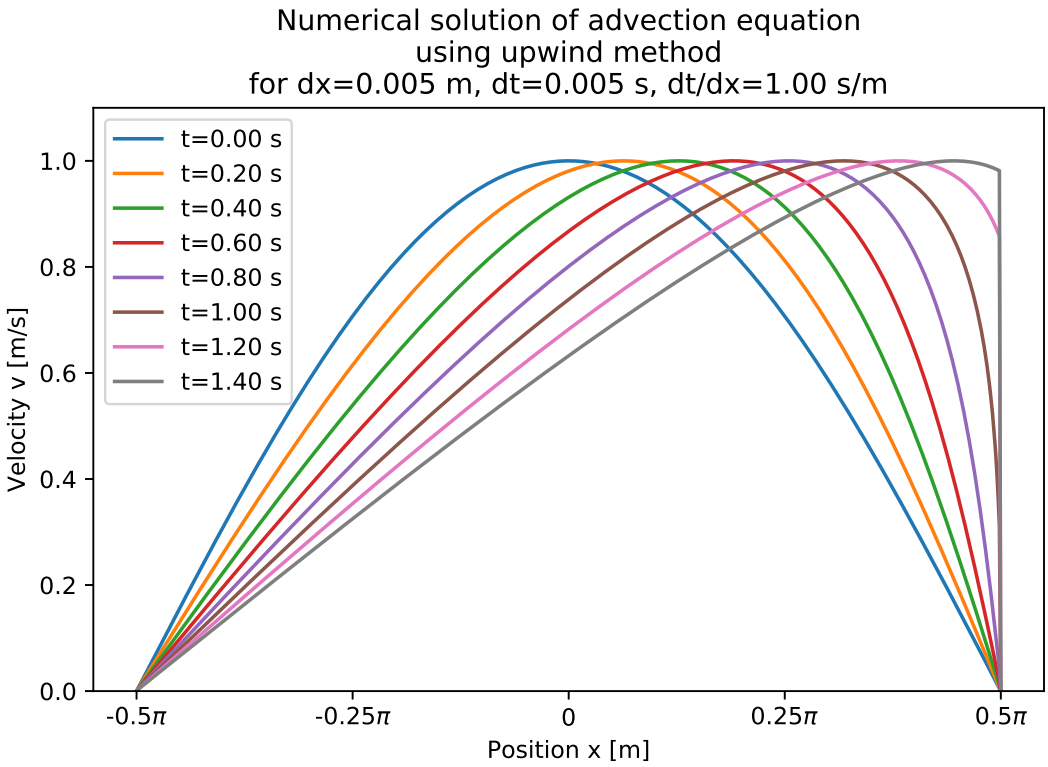


Figure 10: Numerical solution of advection equation calculated with very method using $\Delta t/\Delta x = 1.0\ s/m$. The method appears to be stable.

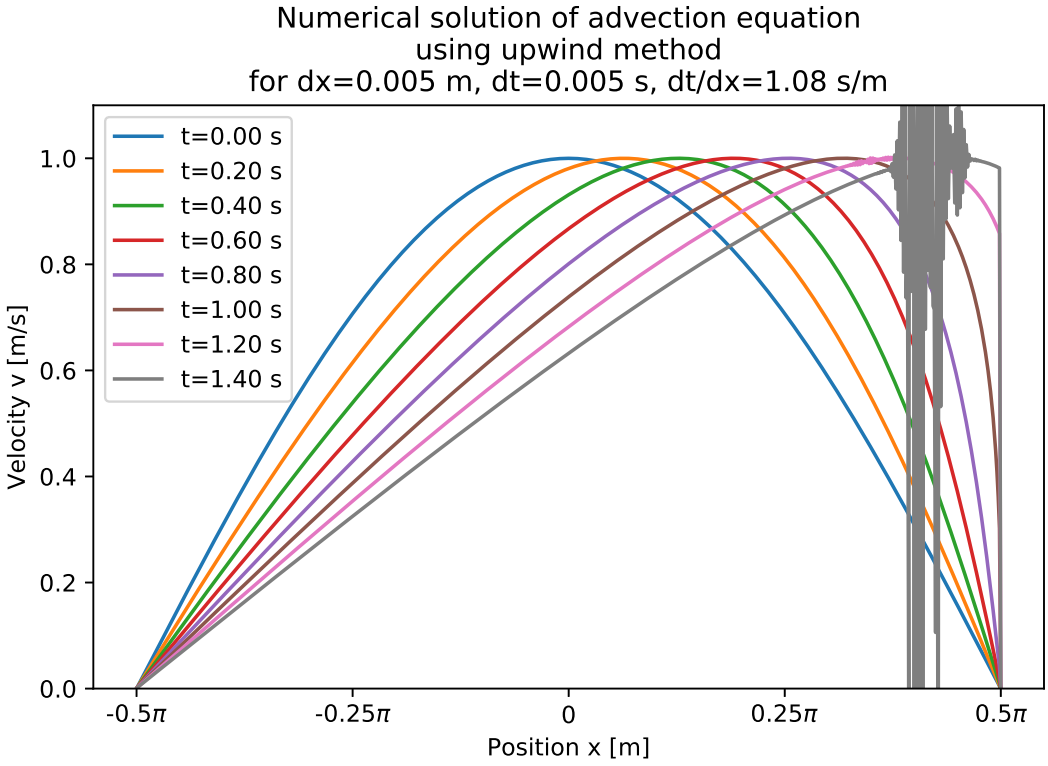


Figure 11: Numerical solution of advection equation calculated with upwind method using $\Delta t/\Delta x = 1.08\ s/m$. Small jiggleschook can be seen at $t = 1.2\,s$, becoming larger at $t = 1.4\,s$.

We can see from Figure 10 that the upwind method appears to be stable at $\Delta t/\Delta x = 1\,s/m$. However, at $\Delta t/\Delta x = 1.08\,s/m$ (Figure 11) the upwind method no longer produces jiggleschnook-free solutions.

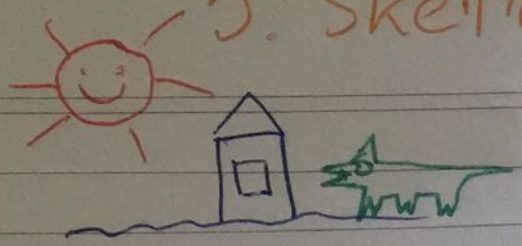
We have found that our implementation of the upwind did not show signs of numerical instabilities for some of the position and time steps we tried given that $\Delta t/\Delta x \leq 1$. Thus, we conclude that our implementation of the upwind method produces more realistic solutions of Equation 1 than our implementation of the centered-difference method, because the former was stable for $\Delta t/\Delta x \leq 1$ and the latter was unstable for large t and all values of $\Delta t/\Delta x$ that we tried.

4 Testing, running the programs and making plots

There are two completely separate source codes for the analytical and numerical calculations. The source code for finding the analytical solution is located in the `01_root_finder` directory, while the numerical part is implemented inside the `02_numerical` directory.

Instructions for compiling, running the program, the units tests, and plotting the results are located in the README.md files that come with the source codes of the programs.

5. SKETCHING BLOBS!



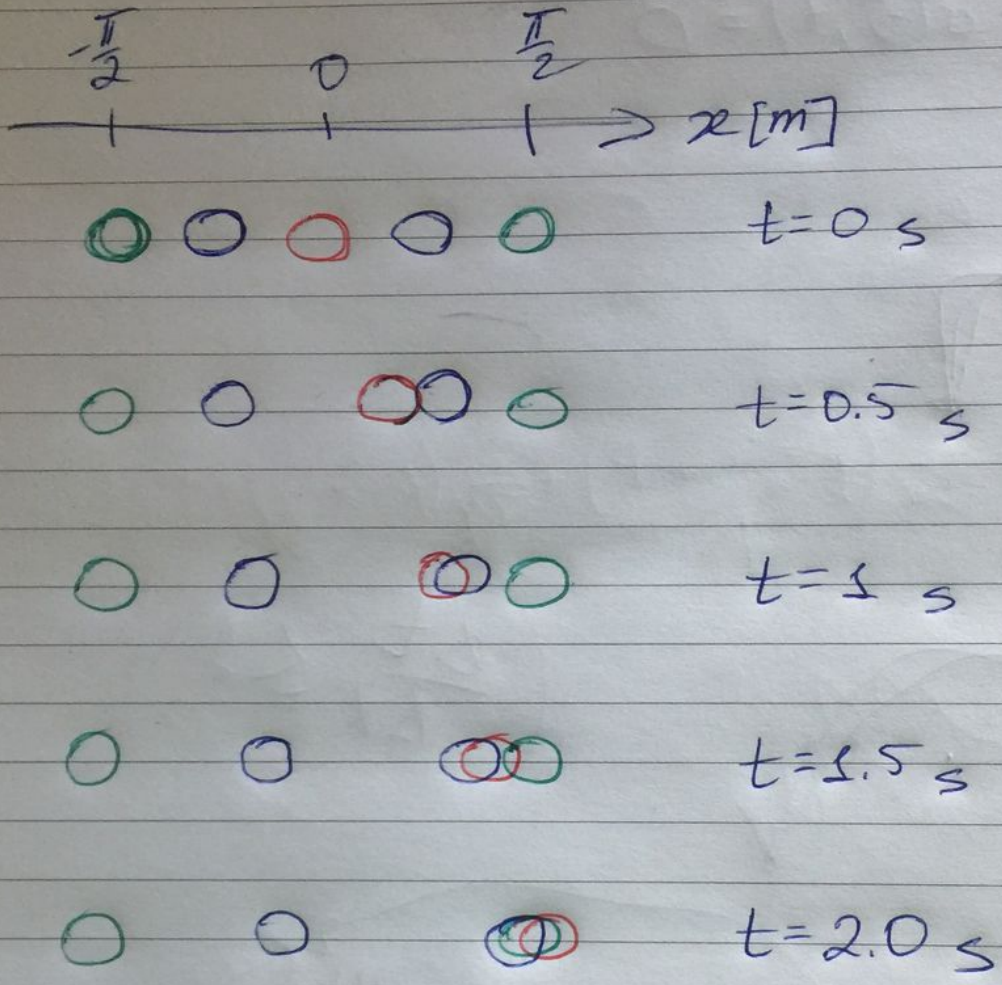


Figure 170: Blobs moving in fluid. Green blob has zero initial velocity. Red blob has initial velocity $1\ \frac{m}{s}$, Blue blob has initial velocity between $0\ \frac{m}{s}$ and $1\ m\ s^{-1}$.